# An Activity Description Language for Activity Recognition

Ines SARRAY*, Annie RESSOUCHE*, Sabine MOISAN*, Jean-Paul RIGAULT* and Daniel GAFFE**
*Université Côte d'Azur, INRIA Sophia Antipolis, France
{ines.sarray, annie.ressouche, sabine.moisan, jean-paul.rigault}@inria.fr

**Université Côte d'Azur, CNRS, LEAT, Sophia Antipolis, France
daniel.gaffe@unice.fr

***Abstract*—Activity recognition aims at recognizing and understanding the movements, actions, and objectives of mobile objects. These objects can be humans, animals, or simple artefacts. Many important and critical applications such as surveillance or health care require some form of (human) activity recognition. Existing languages can be used to describe models of activities, but they are difficult to master by non computer scientists (ex: doctors). In this paper, we present a new language dedicated to end users, to describe their activities. We call it ADeL (Activity Description Language). This language is intended to be part of a complete recognition system. Such a system has to be real time, reactive, correct, and dependable. We choose the synchronous approach because it respects these characteristics, it ensures determinism and safe parallel composition, and it allows verification of systems using model-checking. Relying on the synchronous approach, we supply our language with two complementary formal semantics and we provide it with two formats: textual and graphical. This paper focuses on the description of the ADeL language.**

**Keywords:** Activity Recognition, Activity Description Language, Synchronous Approach, semantics

## I. Introduction

Activities of a person may give an idea about his/her intention and behavior. That is why activity recognition is getting more and more important in all fields, especially in security, monitoring, and healthcare domains. Activity recognition aims at recognizing a sequence of actions that follows a predefined model. Information needed for activity recognition are treated in two steps: First, the data sent by different sensors (video-cameras, audio, binary, analog sensors..) are treated and refined to recognize and track objects, and to detect low-level events. Second, this low-level information is collected and transformed into inputs to an activity recognition engine. We consider these inputs as "primitive events" or situations (example: a person is sleeping, eating, watching TV...).

We propose to model activities using finite automata because it is a "natural" way to describe an activity. There are several possible models of automata, and as we work with real-time and reactive systems, we opted for synchronous finite automata. This type of finite automata allows our recognition system to benefit from the formal foundations of the synchronous approach and automata theory by automatic proofs, static verification, powerful simulation, code generation, etc.

Synchronous automata can be represented by states and transitions or described using a specific language that defines these states and transitions in an *implicit* way. There are different synchronous languages such as Scade, Esterel, Signal, or Lustre [4]. These languages are for expert users. We propose another language that is easier to understand and to use by non computer scientists (e.g., doctors). We call it ADeL (Activity Description Language). Based on the activity models described using ADeL, we aim at building a generic activity recognition system.

The paper is organized as follows. We first give an overview of our general recognition system in section II. Then, in the next section, we present a short reminder of the synchronous model of reactive systems. Section IV is the core of the paper: it focuses on the description of our activity description language and we illustrate it through a use case in section V. Finally we present several related works before concluding.

## II. Global view of the Recognition System

Building a complete generic recognition system involves many different aspects. We give the flavor of each of them in Fig. 1. This figure presents the global structure of our generic activity recognition system. There are two main parts: a *configuration*, performed off line and a *run time* execution.

### A. System Configuration

The first step is to describe the activities to be recognized in our dedicated language, ADeL. These activities are independent and each one corresponds to a self contained ADeL program. In fact, each program is a generic description of an activity in term of abstract *roles* instead of the real *actors* detected at run time. Second, each ADeL program is separately compiled, ultimately producing a (C++) shared library. The running system will load these libraries at run time. Hence the Recognition Engine is able to recognize several activities corresponding to different predefined activity models.

In addition, the ADeL compiler can generate other output formats for simulation, and interface with static analysis tools such as model checkers.

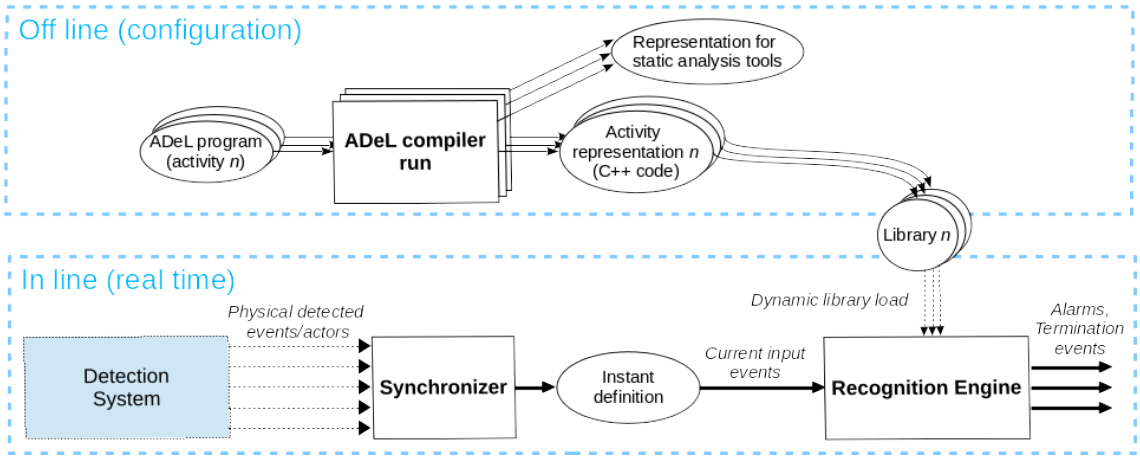


Fig. 1. General structure of our activity recognition system (ADeL is used during the offline configuration step).

### B. Run Time Recognition

The Detection System continuously extracts events and objects (actors) from physical sensor information (e.g., video, audio). These events correspond to the basic actions usable in activity description. It is the role of detection system to take into account possible uncertainty (indeed adressed by most vision systems), so that only "reasonably certain" events are sent to the recognition engine.

The Synchronizer then filters these physical asynchronous events and aggregates them into a logical instant (section VII).

Finally, the Recognition Engine recognizes at run time all effective activities corresponding to at least one model. At each instant, it dispatches input events toward concerned activities, it assigns actors to roles, and possibly creates new instances of activities. It then triggers the transitions of all the ongoing activities and collects the output events (in our case alarms or activity terminations).

## III. Synchronous Model of Reactive Systems

### A. Introduction to synchronous paradigm

Activity recognition systems are reactive, they listen to input events coming from their environment and react to them by generating output events towards the environment. Finite automata are well adapted to the representation of such systems. Recognition systems have to be real time, reactive, correct, and dependable. These stringent requirements justify the use of synchronous model because it allows to describe, analyze, and verify these systems: It ensures determinism and supports concurrency through deterministic parallel composition. In particular, critical races are detected by static analysis and by model checking techniques.

Moreover, synchronous models are a good solution to reduce the complexity of such systems, by considering the system evolution along successive discrete *instants*. Indeed, the Synchronous Paradigm relies on a discrete logical time composed of a sequence of logical instants defined by the system reactions.

An instant starts when input events occur. Internal and output events are computed using input events until stability (fixed point) is achieved. The instant finishes by delivering the output events to the environment. Inputs that come "during" the instant are frozen and not considered. Hence, instants are atomic and their sequence defines the logical time. In this model, instants are considered as discrete, they take “no time” with respect to the logical time they define.

### B. Synchronous Finite Automata

Synchronous models can be represented explicitly as *finite automata*. However, another form has been introduced by G. Mealy [8] to represent sequential efficient circuits as Boolean equation systems computing both the output event values and the next states from input event values and current states. We call this representation "implicit" finite automata.

## IV. Activity Recognition Language (ADeL)

ADeL is a language that allows non-computer scientists to describe their activities. It is a modular and hierarchical language, which means that activities can be simple or composed of one or more sub-activities. ADeL has the notions of (typed) roles, events and sub-activities, flow of control... It supports parallelism, variants (choices), and repetitions.

ADeL relies on formally specified control and temporal operators (detailed in Table I). The operators deal with events coming from the environment. Some of them are instantaneous (**nothing, alert**) and others take at least one instant to process. ADeL facilitates the consideration of real clock time thanks to his two operators **"timeout"** and **"during"**, compared to other synchronous languages where it is too difficult and even impossible for some synchronous languages to treat the real clock time. For example to treat deadlines with the operator "timeout" we express it as follows: $p$ **timeout** $S$ $\{p_1\}$ ($S$ is a timed signal). We express it with Esterel as follows:

```
abort
{p} when S;
present S then alert
else p1;
```

| | |
|---|---|
| **nothing** | does nothing and terminates instantaneously. |
| [**wait**] S | waits for event $S$ and suspends the execution of the scenario until $S$ is present. Operator wait can be implicit or explicit. |
| $p_1$ **then** $p_2$ | starts when $p_1$ starts; $p_2$ starts when $p_1$ ends; the sequence terminates when $p_2$ does. |
| $p_1$ **parallel** $p_2$ | starts when $p_1$ or $p_2$ start; ends when both have terminated. |
| $p_1$ **during** $p_2$ | $p_1$ starts only after $p_2$ start and must finish before $p_2$ end. |
| **while** *condition* $\{p\}$ | $p$ is executed only if the *condition* is verified. When $p$ ends, the loop restarts until the *condition* holds. |
| **stop** $\{p\}$ **when** $S$ [**alert** $S_1$] | executes $p$ to termination as long as $S$ is absent, otherwise when $S$ is present, aborts $p$, sends an alert $S_1$, and terminates. |
| **if** *condition* **then** $p_1$[ **else** $p_2$] | executes $p_1$ if *condition* holds, otherwise executes $p_2$. |
| $p$ **timeout** $S$ $\{p_1\}$ [**alert** $S_1$] | executes $p$ ; stops if $S$ occurs before $p$ terminates and possibly sends alert $S_1$; otherwise executes $p_1$ when $p$ has terminated. |
| **alert** $S$ | raises an alert. |
| **local**$(events)$ $\{p\}$ | declares internal events to communicate between sub parts of $p$. |
| **call**$(scenario)$ | calls a sub-scenario. |

TABLE I
ADEL OPERATORS. $S, S_1$ ARE EVENTS (RECEIVED OR EMITTED); $p$, $p_1$ AND $p_2$ ARE INSTRUCTIONS; *condition* IS EITHER AN EVENT OR A BOOLEAN COMBINATION OF EVENT PRESENCE/ABSENCE.

This part of code seems easy for a programmer but it is not the case for non computer scientists such as doctors. Indeed, it would be even more difficult to represent it in a declarative synchronous language like Lustre. Moreover, it is more complex to use these languages to express the "during" operator.

### A. Formats

We provide our language with both a graphical and textual format.

*1) Graphical Format:* We propose a graphical tool which contains several windows. The most important ones are those for zone, roles, and equipment declaration, and for the description of the activity. In the declaration window, users define the zone of the activity, then select the roles of its actors and the needed equipment. In the activity description window, users specify expected events that participate in the activity and organize them in a story board, in the form of a timelined "organigram", using a tool panel that displays ADeL operators, sub-activities, and events. Moreover, users can describe their activities in a hierarchical and modular way, which means that they can create an activity that includes several sub-activities. This makes the description easier to read and to understand. However, it may be difficult to express complex activities using a purely graphical tool. Thus, we also created a textual format of the language which allows users to describe such activities.

*2) Textual Format:* The description of activities in the textual format consists of several parts: first, define the types and roles of actors; second, define the initial state and expected events that participate in the progress of the activity (we call these expected events sub-activities). Finally, in the body, the user describes the expected behavior of the activity and can build complex instructions by compositions and combinations of operators and events.

### B. Semantics

To provide the language with sound foundations we turn to a formal semantic approach. We supply our language with two complementary semantics. First, a behavioral semantics gives a reference definition of the program behavior using rewriting rules. This semantics is a natural way to describe the behavior of each operator and thus gives them a clear interpretation. Second, an equational semantics describes the behavior in a constructive way and can be directly implemented. This latter semantics transforms a program into a Boolean equation system which represents its automata. Using this equation system ADeL can generate an efficient code for multiple tool targets such as model checkers, syntactic analyzers, etc. Since we have two different semantics, it is mandatory to establish their relationship. In fact we have proved that the execution of a program based on the equational semantics also conforms to the behavioral semantics (for more details see [11]).

## V. USE CASE

We propose a use case in the domain of serious games. Serious games are not only for entertainment but also for teaching, learning, training, communication, or information. Nowadays, they represent an important source of interest in many fields, such as healthcare. Doctors start relying on this kind of games to test patients having cognitive problems, such as Alzheimer or autistic persons.

In this use case, we describe the activity of a serious game to evaluate the behavior and interaction of Alzheimer people. This game consists in displaying on a touchpad a list of different pictures, presenting a random picture in the center, and asking the patient to choose the matching picture in the list. The role of recognition here is to store in a log file information about the patient performance (delays, errors,...) so that doctors can have a record of the patients evolution.

If the patient chooses the right picture, a happy smiley is displayed. An alert saying that the patient did well in the game is sent to the log file and the game displays another picture. Otherwise, a sad smiley is displayed, and an alert saying that the patient chose a wrong picture is sent to the log file and the game asks the patient to try again. If the patient does not interact quickly enough with the game, the game should ask again to choose the right picture from the list. If the patient does not answer before 5 minutes, an alert "game_cancelled" is sent and the game is canceled. If the patient exits the game zone, an alert "test_failed" is sent and the game is aborted.

In the graphical format, users declare roles of actors in the declaration window of the graphical tool(see Fig. 2).

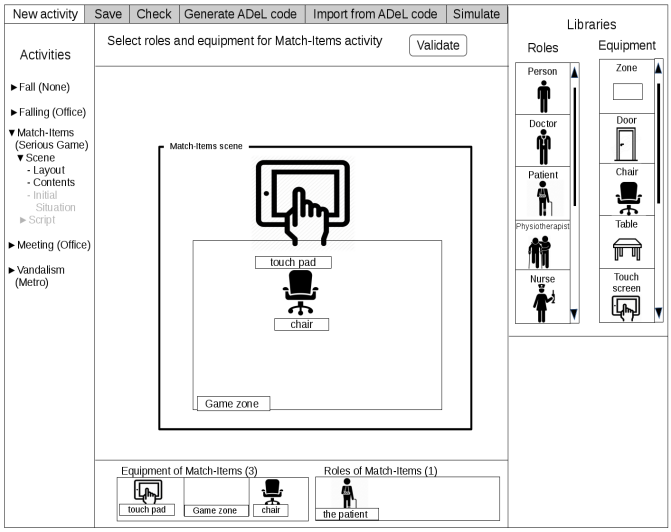


Fig. 2. Graphical format of a serious game description (selection of roles and equipment)

In the activity description window, they declare sub-activities, and describe the steps of their activity along a timeline (see Fig. 3).

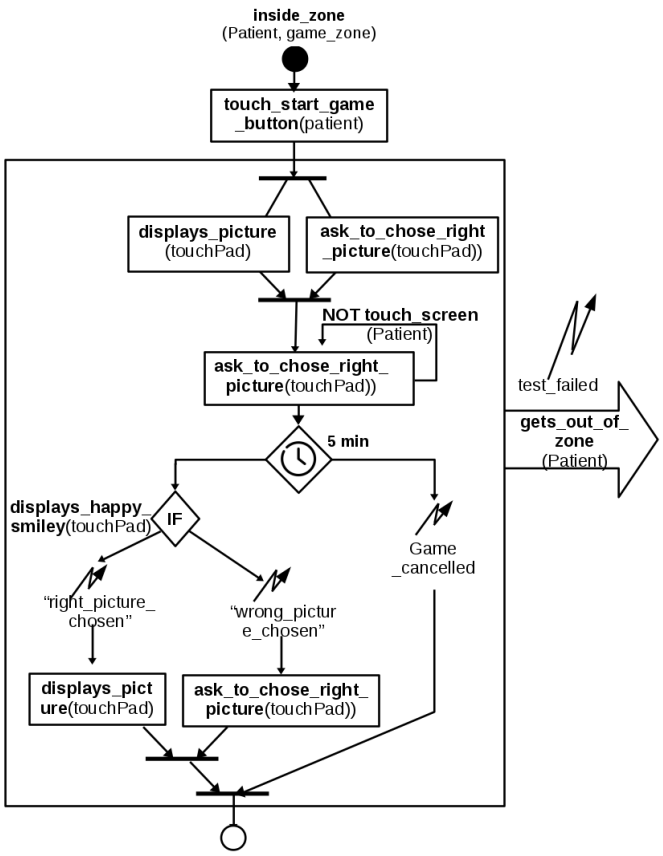


Fig. 3. Graphical format of the serious game description

In the textual format of ADeL, users first declare types of actors. For this use case, there are 3 types: a Zone where the activity takes place, a Person, and some Equipment.

Then, they have to attribute roles to actors: in our case, we need a patient (Person), a touchpad (Equipment) and a game_zone (Zone). The declaration is as follows:

```
Types
Person, Equipment, Zone;
Roles
patient:Person;
touchPad:Equipment;
game_zone:Zone;
```

After that, users define the name of the activity, its expected events and sub-activities:

```
Activity seriousGame:
SubActivities
   touch_start_game_button(patient);
   displays_picture(touchPad);
   ask_to_chose_right_picture(touchPad);
   touch_the_screen(patient);
   displays_happy_smiley(touchPad);
   displays_sad_smiley(touchPad);
   gets_out_of_zone(patient, gameZone);
```

Finally, they describe the serious game activity by defining the initial state at the beginning, and by combining the sub-activities using operators of the language.

```
InitialState : inside_zone(patient, touchPad, game_zone)
Start
touch_start_game_button(patient)
then
  stop when gets_out_of_zone(patient, game_zone)
  {
    displays_picture(touchPad)
      parallel
    ask_to_chose_right_picture(touchPad)
    then
    while NOT touch_screen(patient)
    {ask_to_chose_right_picture(touchPad)}
          timeout 5.0 min
          {
            if  displays_happy_smiley(touchPad)
            then
            {
            Alert("right_picture_chosen")
            then
            displays_picture(touchPad)
            }
            else
            {
            Alert("wrong_picture_chosen")
            then
            ask_to_chose_right_picture(touchPad)
            }
          }
          alert game_cancelled
  }
  alert test_failed
End
```

## VI. Compilation and Validation

We rely on the equational semantics to compile ADeL programs. Based on this semantics, we first transform the program into an equation system representing the synchronous automaton. Then we implement directly this equation system and we transform it into a Boolean equation system.

These equations may be dependent on each other. Thus, we need to find a valid order (compatible with their inter-dependencies) to be able to generate code for execution, simulation, and verification. We defined an efficient sorting algorithm [10], using a critical path scheduling approach,which computes all the valid partial orders instead of one unique total order. This facilitates the merging of several equation systems, permitting incremental compilation: we can include an already compiled and sorted code of a sub-activity into a main one without any recompilation. Thanks to equational semantics rules, we generate a compact and efficient representation in the form of BDDs, thus the ADeL compilation is straightforward, fast, and with a negligible cost [11].

Boolean equation representation of systems also allows the verification and validation of ADeL programs, by generating a format suitable for model checkers such as the off-the-shelf NuSMV model-checker[1]. For instance, for this use case, we proved that the alert "game_cancelled" is sent when the timeout is over.

## VII. Synchronizer

As the world is not synchronous and since we are working with the synchronous paradigm, we have to face the classical problem of sampling. Our system has to deal with asynchronous events that come from the environment. We propose a synchronous transformer, we call it Synchronizer. As shown in section II, the Synchronizer receives asynchronous events from the environment, filters them, decides which ones may be considered as “simultaneous”, and groups them into a logical instant according to predefined policies. The sequence of these instants constitutes the logical time of the recognition engine. In general, no exact simultaneity decision algorithm exists but several empirical strategies may be used for determining instant boundaries, relying on event frequency, event occurrence, elapsed time, etc. Since we are using temporal operators, we should also manage the clock time. Thus, in case of temporal operators, the Synchronizer considers the clock time as an event like others and sends it to the recognition engine.

## VIII. Related work

Synchronous languages such as Esterel [4] are meant to describe reactive systems in general and thus can be used to describe human activities. These languages, like ADeL, use a *logical* time which means that the recognition is performed only when something meaningful occurs. Although their syntax is rather simple, their large spectrum makes them difficult to master by some end users. Being dedicated to activity description, a language like ADeL appears more "natural" for its end users. To improve its acceptance and its ease of use by non computer scientists, we are developing the graphical interface with ergonomists (LUDOTIC) and we collaborate with doctors from Claude Pompidou hospital. All these synchronous languages have been given formal semantics. For instance, Esterel has several semantics, with different purposes. In particular, one of these semantics provides a direct implementation under the form of “circuits”. ADeL adopts a similar approach more suited to our purpose. It also simplifies some operators whose semantics in Esterel is complex.

Message Sequence charts [3], [1], which are now introduced in UML, and Live Sequence Charts [7] are also specification languages for scenarios with a graphical layout that immediately gives an intuitive understanding of the intended system behavior. As detailed in [3], [1], Message Sequence Charts (MSCs) is a popular specification language to describe activities and system behaviors via simple graphical representations based on lifelines and exchanged messages between communicating entities. It is also possible to model these activities using MSCs finite state automata. This representation is called High-level Message Sequence Charts (HMSCs) and it supports parallel composition. The hierarchical composition of HMSCs and the MSC operators are similar to our approach and compatible with ADeL features.

However, in [2] researchers detected some pathologies in MSCs: we can have incorrect behaviors because of some message interaction pattern and MSCs specifications defects. These pathologies mainly affect synchronization issues. For instance, races may occur from conflicts between visual order defined by the semantics and system causalities. The synchronous paradigm can deal with these synchronization problems. In our case, race conditions are detected at compile time and the program is rejected. Another pathology comes from possible ambiguous choices between events, because of the denied access to other processes. Thanks to the synchronous approach, we avoid this ambiguity and control problems by relying on solid concepts such as determinism. A solution to avoid problems mentioned above with MSCs is model checking and formal verification. In [1] researchers illustrate problems of the MSCs models verification for synchronous and asynchronous interpretations and suggest different techniques to solve these model checking problems such as using the automata-theoretic approach or defining algorithms to check boundedness. In our case, using the Boolean equations of the program, we can interface automatically with model checkers such as NuSMV, but it is not mandatory because in the synchronous approach, most of these pathologies are compile-time checked.

Another scenario-based specification and modeling language is Live Sequence Charts(LSCs) [7]. It represents an extension of MSCs and UML2 sequence diagrams but it is more expressive and semantically richer, which makes it useful in different steps of software development and verification process, and for all kinds of applications such as Web ones [7]. It has a formal semantics allowing analysis and verification. Model checking of LSCs is possible by translating them into temporal logic, but the size of the resulting formula, even for simple LSCs, makes model checking difficult. However, [6] proposes a more efficient translation, but only for a class of LSCs. Similarly to ADeL, LSCs models are used to specify the behavior of either sequential or parallel systems. They can also be transformed to automata, which helps researchers in [7]

[1] http://nusmv.fbk.eu/

to verify and test scenarios using a depth-first search method.

In [12], authors propose a natural and intuitive language to describe scenario models using actors, sub-scenarios, and constraints. They also introduce temporal constraint resolution techniques to recognize scenarios in real time. This approach is tied to real time video interpretation. For instance, it is subject to a unique physical time base, namely the frame rate. Thus the same event can be detected along several consecutive frames, possibly overwhelming the recognition process. Our use of logical time tends to avoid this kind of problem. Moreover, we aim at a generic approach that can be used in several domains, independently of the nature of events and sensors.

In [9] the authors address activity recognition in smart homes to provide assistance in Activities of Daily Living. Their recognition objective is similar to ours and they refer to the same concepts: activity model with variants, decomposition into sub-activities, temporal relations, etc. Their description of activities uses ontologies to model the relations between activities and their involved entities (actors, location, resource...). They combine these ontologies with temporal knowledge representation based on Allen's algebra. In our case, we do not use ontologies but classical typing and hierarchical decomposition of activities. Compared to their declarative approach, we model temporal relations in a more imperative way, since ADeL itself is an imperative language.

Authors in [5] are also working on recognizing human activity on-line in smart homes, based on streaming sensor data. These streaming data are treated according to the sliding window based approach which consists in dividing each sensor data stream into windows that contain an equal number of sensor events. As different activities detected by sensors may be described by different window lengths of sensor events, researchers in this work also consider other parameters for data processing, such as the time dependancy, information based weighting of sensor events within a window, and past contextual information. In our case, we can adopt a part of this approach for the treatement of data coming from sensors, but we use also other policies. Our activity recognition system is synchronous, thus, this part of the work is taking place in the Synchronizer.

## IX. Conclusion

In this paper, we presented a new programming language for human activity description, called ADeL.

The description of activities using ADeL and the generation of their corresponding automata is a part of a complete recognition system. This recognition system is a run-time system that aims to recognize activities based on the matching between these automata and events coming from different sensors in the environment.

We chose to rely on the synchronous approach because it ensures the concurrency management, the generation of deterministic programs, verification through model checking facilities, and it has a well-established formal foundation. It also allows us to make verification and validation of the system by generating code for model-checkers.

We also defined two formal semantics for ADeL: one of them provides an abstract description of the program behavior and the other helps us to compile this program into an automaton described as an efficient equation system.

As future work, in collaboration with ergonomists, our first project is to improve and simplify the ADeL language to be easier to use by non computer scientists.

Our primary tests have proved the ease of integration of the code generated by ADeL (basically composed of Boolean) in the recognition system and its efficiency at run time. Concerning the asynchronous to synchronous transformation, we still have no exact solution, but we are working on several strategies and heuristics to constitute synchronous instants. Large scale experiments are still necessary.

## References


[1] Rajeev Alur and Mihalis Yannakakis. *Model Checking of Message Sequence Charts*, pages 114–129. Berlin, Heidelberg, 1999.

[2] Haitao Dan, Robert M. Hierons, and Steve Counsell. A framework for pathologies of Message Sequence Charts. *Inf. Softw. Technol.*, 54(11):1283–1295, nov 2012.

[3] Thomas Gazagnaire, Blaise Genest, Loïc Hélouët, P. S. Thiagarajan, Shaofa Yang, and Vasco T. Vasconcelos. *Causal Message Sequence Charts*, pages 166–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[4] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[5] Narayanan C. Krishnan and Diane J. Cook. Activity recognition on streaming sensor data. *Pervasive Mob. Comput.*, 10:138–154, February 2014.

[6] Rahul Kumar, Eric G. Mercer, and Annette Bunker. Improving translation of Live Sequence Charts to temporal logic. *Electron. Notes Theor. Comput. Sci.*, 250(1):137–152, September 2009.

[7] L. Li, H. Gao, and T. Shan. An executable model and testing for Web software based on Live Sequence Charts. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, June 2016.

[8] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Sys. Tech. Journal*, 34:1045–1080, September 1955.

[9] George Okeyo, Liming Chen, and Hui Wang. Combining ontological and temporal formalisms for composite activity modelling and recognition in smart homes. *Future Generation Computer Systems*, 39:29 – 43, 2014. Special Issue on Ubiquitous Computing and Future Communication Systems.

[10] Annie Ressouche and Daniel Gaffé. Compilation modulaire d'un langage synchrone. *Revue des sciences et technologies de l'information, série Théorie et Science Informatique*, 4(30):441–471, June 2011.

[11] Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, and Daniel Gaffé. Synchronous Automata For Activity Recognition. Research report, Inria Sophia Antipolis, April 2017.

[12] Van-Thinh Vu, Francois Bremond, and Monique Thonnat. Automatic video interpretation: A novel algorithm for temporal scenario recognition. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1295–1300, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.